EE 341 – Lab 4

Digital Filtering and Audio Equalizer

Lab Objectives

In this lab, we will experiment with two types of digital linear time-invariant (LTI) filters: finite impulse response (FIR) filters and infinite impulse response (IIR) filters. We will inspect the characteristics of the filters in both time and frequency domain. By completing this lab, you will gain insights into the properties of different types of digital filters and gain familiarity with the designs and implementations of them.

Part 1: Digital Filters

To start, create a new Jupyter Notebook in your ee341lab folder and name it to lab4. Import the modules: `numpy`, `pyplot`, `fftpack`, and `simpleaudio` just as you did in Lab 3. Also import the `signal` module from SciPy library as follows.

```
from scipy import signal
```

The `signal` module provides several convenient functions for creating filters and applying them to signals. It will be our main tool for implementing filters in this lab.

Then, download `microsoft_stock.txt` and `music.wav` from Canvas and copy them to the same directory where your Jupyter Notebook is stored. We will use the signals from these files to test our filters.

Background

Every casual LTI digital filter can be described by a linear difference equation that relates the output of the filter $y[n]$ to the input of the filter $x[n]$:

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

The time-domain difference equation presented above is equivalent to the frequency-domain transfer function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{\sum_{k=0}^{N} a_k z^{-k}} \; where \; z = e^{j\omega}$$

In the equation, a_k and b_k are the filter coefficients. a_k is an array of length $N+1$ and b_k is an array of length $M+1$. The length of the filter, also called the number of taps, is the maximum of the lengths of the two coefficient arrays, i.e. $\max(N, M) + 1$. The order of the filter is the length of the filter minus 1, i.e. $\max(\mathrm{N}, \mathrm{M})$.

By rearranging the equation, we can isolate the current output $y[n]$ to the left-hand side:

$$y[n] = \frac{1}{a_0}\left(\sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]\right)$$

This gives us the formula for calculating the current output of the filter from the current input, past inputs and past outputs of the filter. The leading denominator coefficient a_0 is usually set to 1, since the effect of a non-unit a_0 is equivalent to dividing all other coefficients by a_0.

In Python, LTI digital filters can be implemented using the `lfilter` function from the `signal` module:

```
y = signal.lfilter(b, a, x)
```

In the example code above, `x` is the input signal, `y` is the output signal, and `a` and `b` are the filter coefficients. All of these 4 are stored as 1D ndarrays.

Types of Filters

Filters can be classified by the shape of their frequency responses, which determine their applications.

- **Low-pass filter** removes high-frequency components of the data, allowing only frequencies lower than the cut-off frequency to pass through. It is commonly used for smoothing the data. For audio, applying low-pass filter removes high-pitched sound and upper harmonics, which results in a warm and muddy sound. For image, applying low-pass filter removes sharp details, which results in a blurred image.
- **High-pass filter** removes low-frequency components of the data, allowing only frequencies higher than the cut-off frequency to pass through. It is commonly used for removing DC component from the signal and for extracting details from the data. For audio, applying high-pass filter removes the bass component, which results in a sharp and crispy sound.
- **Band-pass filter** allows only a certain range of frequency to pass through. It is usually used in communication systems, such as radio, to select a channel. Multiple band-pass filters can also form an equalizer which offers detailed control of the level of each band.
- **Band-reject filter** removes a certain range of frequency. It is usually used to remove a specific unwanted frequency from the signal, e.g. removing the feedback of a loudspeaker.
- **All-pass filter** lets all frequency components equally pass through. However, it applies different amount of phase-shift (delay) to different frequency, which is useful for some applications.

FIR Filter

LTI digital filters can also be classified into two types by structure: finite impulse response (FIR) filter, and infinite impulse response (IIR) filter. A FIR filter is a filter with no denominator coefficients a_k, except for a_0, which we set to 1. Therefore, by definition, the output of a FIR filter can be expressed as:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

Shown by the equation, the output of a FIR filter is simply a weighted average of the current and past inputs, with the weights being the nominator coefficients b_k. For this reason, people also call it as the moving average filter. Also notice that the equation exactly describes the convolution operation: $y[n] = b[n] \circledast x[n]$, with the impulse response being the nominator coefficients b_k. Because there can only be a finite number of nominator coefficients, the length of the impulse response is also finite; hence the name “finite impulse response”.

Since the impulse response has finite length, you can implement the filter by directly computing the convolution instead of using `signal.lfilter`. To calculate convolution, use `signal.convolve` as follows:

```
y = signal.convolve(x, h)
```

where `h` is the impulse response of the filter, stored as a 1D ndarray. The advantage of `convolve` is that it can leverage the circular convolution theorem to use FFT to speed-up the calculation.

Task 1: Implement a Moving Average Filter

Load the data in `signal_data.txt` to an ndarray using the following statement:

```
data = np.genfromtxt('signal_data.txt')
```

The data is a 100-second sinusoidal signal with random noise. Plot the data. Notice the quick changes in the data which correspond to the high frequency component. People tend to remove the noise by using a moving average filter. Consider the following 30-second moving average filter:

$$y[n] = \frac{1}{30}(x[n] + x[n-1] + \cdots + x[n-29])$$

which corresponds to the impulse response:

$$h[n] = \frac{1}{30}(u[n] - u[n-30])$$

Apply this filter to the 100-second signal using both `lfilter` and `convolve`. For each the two implementations, plot the original data and the filtered data in the same plot. Does the filtered result look smoother? Plot the magnitude of the frequency response of the filter to verify that this system corresponds to a low-pass filter.

In your notebook, include the plot for the `lfilter` implementation, the plot for the `convolve` implementation and the plot of the system frequency response. Be sure to label the axes and give each plot a meaningful title. Explain the differences in the results of the two implementations. How would you adjust the implementations so that they give identical results?

IIR Filter

We already know that FIR filters don't have any denominator coefficients a_k for $k \geq 1$. IIR filters, in contrast, have some non-trivial denominator coefficients. Therefore, by definition, the output of an IIR filter can be expressed as:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]$$

As shown by the equation, the output of an IIR filter is the weighted average of not only the current and past inputs, but also past outputs. Since the current value of y depends on the previous values of y, IIR filter is also known as the recursive filter. Because of the recursion, the impulse response of a stable IIR filter decays over time but never dies off, making it infinite length, and hence the name "infinite impulse response".

Apart from the length of the impulse response, FIR and IIR have many other different properties as listed below.

- **Order:** FIR filters require a large filter order to meet the requirements of passband, stopband, ripple, and roll-off. In contrast, IIR filters can accomplish them with a lower order. Therefore, FIR filters are computationally more expensive compared to IIR filters.
- **Frequency Response:** The frequency response of a FIR filter has ripples in both passband and stopband and has a slow roll-off at the cut-off frequency. In contrast, the frequency response of an IIR filter is much smoother and has sharper roll-off at the cut-off frequency.
- **Stableness:** The poles of a FIR filter can only be at the origin, so it must be stable. In contrast, the poles of an IIR filter can be anywhere, so it can be either stable or unstable.
- **Phase:** FIR filters can be designed to have a linear phase which preserves the shape of the waveform. In contrast, IIR filters have non-linear components in the phase response that can distort the shape of the waveform.

Using Classic Filter Designs

Instead of designing filters by ourselves, we can use some classic filter designs provided by the SciPy package. For IIR filter, we can use the Butterworth design, which is a design that has no ripple in the frequency response. To create a Butterworth filter, use the `signal.butter` function:[1]

```
b, a = signal.butter(order, W, type)
```

where `b` and `a` are the filter coefficients, `order` is the order of the filter, `freq` is the cutoff frequency and `type` can be either `'lowpass'` or `'highpass'`. The unit of `W` is the normalized frequency, which has the range $0 \leq W \leq 1$ where 1 corresponds to the Nyquist frequency (half of the sampling rate). This range also maps to the DTFT or DFT range: $0 \leq \omega \leq \pi$ or $0 \leq f \leq 0.5$ we used in Lab 3.

For FIR filter, we can use the window-method design, which approximates the impulse response of an ideal filter. To create a window-method filter, use the `signal.firwin` function:

```
# Lowpass:
b = signal.firwin(num_taps, W)
# Highpass:
b = signal.firwin(num_taps, W, pass_zero = False)
```

where `b` is the numerator coefficient or impulse response, `num_taps` is the length of the filter and `W` is the cutoff frequency which has the unit of the normalized frequency.

Task 2: Analyze the Characteristics of Filters

For this task, you will use the `analyze` function provided in Appendix A to analyze the characteristics of 4 different filters. In particular, we will look at how FIR and IIR filters behave differently in terms of frequency response, impulse response and poles / zeros.

To start, copy the `analyze` function in Appendix A to a new code cell and execute it. Then, implement the 4 following filters. Each filter should have an order of 10 and a cutoff frequency of $\omega = 0.1\pi$.

- Low-pass IIR filter using Butterworth design.
- Low-Pass FIR filter using window-method design.

[1] The Butterworth filter is actually an analog (continuous-time) filter, so the digital IIR version is only approximated.

- High-pass IIR filter using Butterworth design.
- High-pass IIR filter using window-method design.

Then for each filter, apply the `analyze` function, which will produce plots of the frequency response, impulse response and poles / zeros.

Finally, apply each filter to the each of the following signals:

- The data in `microsoft_stock.txt`, which is the daily stock price of Microsoft over 4 years.
- A pulse of length 20:
  $x[n] = u[n] - u[n-20]$ where $x[n]$ is of total length 60 (so append 40 zeros to the end).

In your notebook, include the 4 plots produced by the `analyze` function, and for each input signal, plot the original signal and the output of each filter in the same set of axes. Comment on the differences in the frequency response of the two filters (magnitude and phase) and how this impacts the outputs. When commenting on frequency responses, consider how close the filter matches an ideal filter.

Part II: Audio Equalizer

In this section, you will implement an audio equalizer using three different filters. Equalizer works by first breaking down the audio signal into multiple frequency bands using filters. Each band will then receive a different gain. Finally, all bands are summed together to produce the output signal. Audio mixing boards do exactly this, though with many more bands than the three bands we have here. The term "equalizer" comes from the fact that people often want to boost the gain of low-energy frequency ranges of the sound or to reduce the gain of high-energy frequency ranges of the sound. By doing so, each frequency range of the sound will have about equal loudness, allowing them to mix together nicely.

Shown below is the signal flow diagram of the 3-band equalizer you will implement. G_1, G_2 and G_3 are correspondingly the gains of the bass, mid and treble frequency ranges. The coefficients of each filter are provided in Table 1. However, you need to figure out by yourself which filter does each set of coefficients correspond to. The magnitude response of each filter is shown in Figure 2.

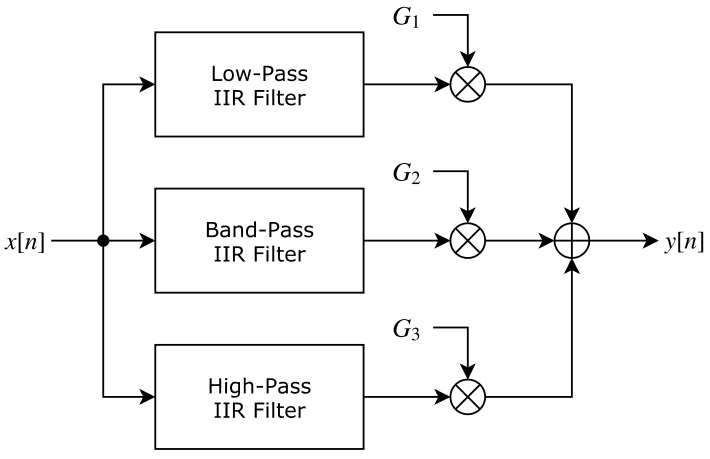

Figure 1: Signal flow of the equalizer

Table 1: Filter coefficients of the equalizer

Filter Coefficients	Which filter? (LP, HP or BP?)
`b1 = [11.713, 0, -23.4257, 0, 11.713]` `a1 = [1800, -6656.7, 9341, -5896.1, 1413.4]`	
`b2 = [0.2982, 0.89471, 0.89471, 0.2982]` `a2 = [1800, -4989.8, 4625.2, -1433]`	
`b3 = [688.1, -2752.5, 4128.71, -2752.47, 688.12]` `a3 = [1000, -3256.6, 4033.77, -2246, 473.51]`	

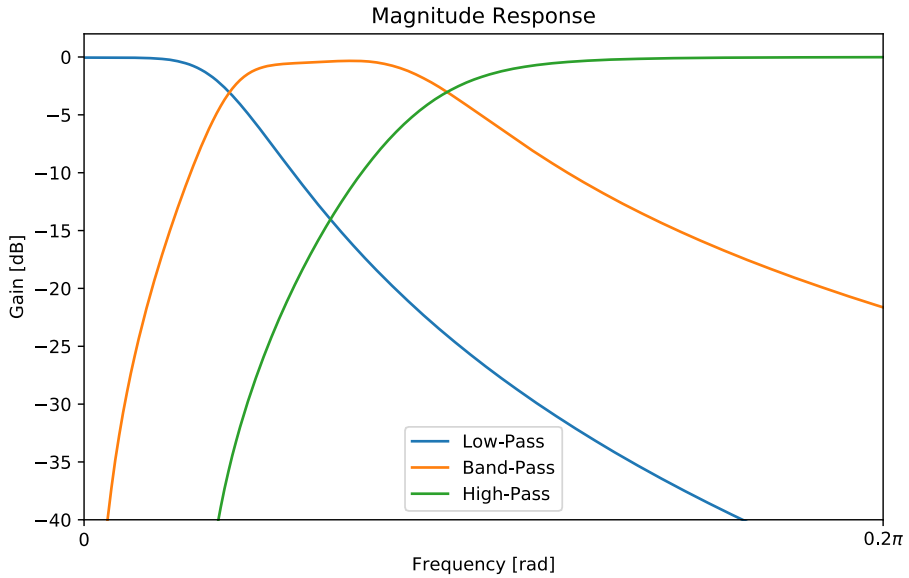

Figure 2: Magnitude response of the equalizer

Task 3: Implement the 3-Band Equalizer

First, identify which filter does each set of coefficients in Table 1 correspond to. Then, implement the equalizer by writing a function that takes a sound signal and the gain terms G_1, G_2 and G_3 as inputs, applies the filters, multiplies the filter outputs by the gain terms, and sums the results. The gain terms supplied to the function should be in decibels (dB).

After you implemented your equalizer, load the sound file `music.wav` using the function provided in Appendix B (the same one you used in Lab 3). Apply the equalizer to the music with $G_1 = G_2 = G_3 = 0\ dB$ and play the output. Verify that it sounds the same as the original input. Then, experiment with different sets of gains (for example $G_1 = G_2 = 0\ dB$ and $G_3 = -40\ dB$, or $G_1 = -40\ dB$ and $G_2 = G_3 = 0\ dB$). Discuss how the filtered sound sounds different in at least two different cases.

To play sound, use the function below.

```
def play(samples, sample_rate):
  sa.play_buffer((np.clip(samples, -1, 1) * 32767).astype('int16'),
                 1, 2, sample_rate).wait_done()
```

In your notebook, include your answers to the blanks in Table 1, the code of your equalizer function and the code of playing the outputs of the equalizer. Discuss how the filtered sound sounds different under different sets of gains.

Developed/revised by Kevin Lin, Yibo Cao, and Tai Chen 2019

Lab Submission Requirements

Submit your completed Jupyter Notebook in the `ipynb` format to Canvas. The notebook file can be obtained by clicking the menu: File – Download as – Notebook or directly accessed from the ee341lab folder in your home directory. Make sure to include the code of all the 3 tasks in code cells and include the required discussions of each task in text cells.

Appendix A: Filter Analysis Function

Copy the function below to a code cell and execute it. To apply it to a filter, call `analyze(b, a, title)` where `b` and `a` are the filter coefficients and `title` is the title of the plots.

```
def analyze(b, a, title):
  b = np.reshape(b, -1)
  a = np.reshape(a, -1)
  fig = plt.figure(figsize = (10, 8))
  fig.suptitle(title, fontsize = 'x-large')
  fig.subplots_adjust(hspace = 0.5, wspace = 0.3)

  # Plot frequency response
  w, h = signal.freqz(b, a, 1024)
  w_majors = np.array([0, 0.2, 0.4, 0.6, 0.8, 1]) * np.pi
  w_labels = ['0', r'$0.2\pi$', r'$0.4\pi$',
              r'$0.6\pi$', r'$0.8\pi$', r'$\pi$']
  ax = fig.add_subplot(221)
  ax.plot(w, 20 * np.log10(np.abs(h)))
  ax.set_xticks(w_majors)
  ax.set_xticklabels(w_labels)
  ax.set_title('Magnitude Response')
  ax.set_xlabel('Frequency [rad]')
  ax.set_ylabel('Gain [dB]')
  ax = fig.add_subplot(222)
  ax.plot(w, np.rad2deg(np.unwrap(np.angle(h))))
  ax.set_xticks(w_majors)
  ax.set_xticklabels(w_labels)
  ax.set_title('Phase Response')
  ax.set_xlabel('Frequency [rad]')
  ax.set_ylabel('Phase [deg]')

  # Plot impulse response
  x = np.zeros(50); x[0] = 1
  y = signal.lfilter(b, a, x)
  ax = fig.add_subplot(223)
  ax.stem(np.arange(len(y)), y)
  ax.set_title('Impulse Response')
  ax.set_xlabel('Time')
  ax.set_ylabel('Output')

  # Plot poles & zeros
  n_taps = max(len(b), len(a))
  b = np.pad(b, (0, n_taps - len(b)), 'constant')
  a = np.pad(a, (0, n_taps - len(a)), 'constant')
  z, p, k = signal.tf2zpk(b, a)
```

Developed/revised by Kevin Lin, Yibo Cao, and Tai Chen 2019

```
  ax = fig.add_subplot(224, projection = 'polar')
  ax.plot(np.angle(p), np.abs(p), 'x')
  ax.plot(np.angle(z), np.abs(z), 'o', markerfacecolor = 'none')
  lines, labels = ax.set_rgrids([1])
  for line in lines: line.set_color('black')
  ax.set_thetagrids([0, 90, 180, 270], ['Re(z)', 'Im(z)'])
  ax.set_rlabel_position(0)
  ax.spines['polar'].set_visible(False)
  ax.set_title('Poles & Zeros', y = 1.1)
```

Appendix B: Loading WAV Files

To load sound data from WAV files into your Python program, firstly copy the files you want to load into the same directory where your Jupyter Notebook is stored (that is the ee341lab folder in your home directory if you followed our convention in Lab 1). Then, add the following import statement after your other imports:

```
import scipy.io.wavfile as wav
```

Ideally, you should be able to directly use `wav.read` from this package to load files. However, WAV files can have different quantization formats and number of channels that will cause troubles later on. Therefore, we provide you the following helper function to help you deal with different formats.

```
# Load a WAV file.
# Return the sampling rate and the sample array.
def wav_load(file_name):
  # Load the raw data.
  sr, data = wav.read(file_name)
  # Only use the first channel.
  if data.ndim > 1:
    data = data[:, 0]
  # Convert to float32 quantization.
  kind = data.dtype.kind
  bits = data.dtype.itemsize * 8
  data = data.astype('float32')
  if kind == 'i' or kind == 'u':
    data = data / 2 ** (bits - 1)
    if kind == 'u':
      data = data - 1
  return sr, data
```

Example usage:

```
sr, samples = wav_load("do.wav")
```

This will load the file `do.wav`, putting the sampling rate into the variable `sr` and putting the sample array into the variable `samples`.